The Gist of Side Effects in Pure Functional Languages

Pablo Nogueira
http://www.google.com/search?q=Pablo+Nogueira

23rd September 2004

Abstract

We explain the gist of how to attain side effects in *pure* functional programming languages via *monads* and *unique types* with input-output as a motivating example. Our vehicle for illustration is the strongly type-checked, pure, and non-strict functional language Haskell. The category-theoretical origins of monads are explained. Some basic notions of Category Theory are also presented in programming terms. We provide a list of suggested reading material in the references.

FUNCTIONAL PROGRAMMING

Functional programming is based on two central ideas: (1) computation takes place by evaluating applications of functions to arguments and (2) functions are first-class values. In particular, functions are *higher-order* (can be passed to or be returned by other functions) and can be components of data structures. Functional languages differ on whether they are strongly type-checked, weakly type-checked, or untyped; whether they are dynamically type-checked or statically type-checked; whether they are pure or impure; and finally whether they are strict or non-strict.

Pure functional languages lack assignment constructs, an expression produces the same value independently of when it is evaluated—a property called *referential transparency*—, and side-effects like input-output are carefully controlled and separated at the type level by so-called *monads* and *unique types*. They usually have non-strict semantics for functions and their evaluation order is typically *lazy* (*i.e.*, *call-by-need*). In contrast, *impure* functional languages allow side effects like imperative languages, they have strict semantics, and evaluation order is *eager* (*i.e.*, *call-by-value*). (Recently, a new synthesis of call-by-value and call-by-*name*—*i.e.* call-by-need without the sharing and *memoising* [Rea89]—has appeared with the name of *call-by-push-value* [Lev04].)

In the present introduction we explain the gist of how to attain side effects in *pure* functional programming languages. Our vehicle for illustration is the strongly type-checked, pure, and non-strict functional language Haskell[1] which is pretty much the 'standard' lazy language. Purity and non-strictness are not just a matter of style. Programs in impure, strict languages will look and work quite differently than their pure counterparts. The main benefit of purity is referential transparency [BW88]. The main benefits of non-strictness are higher modularity and lower coupling from evaluation concerns [Hug89, Rea89].

PURITY AND EFFECTS

Purity is not at odds with computations that require side effects like input-output, destructive updates, concurrency, exceptions, or language inter-operation. How is this possible? It is well-known that expressions in pure functional languages are referentially transparent so variables are *immutable* or *persistent*—they cannot hold different values at different times. But it is also well-known that stateful computation can be

[1] http://www.haskell.org

'simulated' functionally by passing the state explicitly from function to function as an extra parameter; a mechanism somewhat confusingly called 'threading' because the state is 'threaded' from function to function. (The terms 'threading' and 'thread' have acquired a more specific meaning from their use in concurrency and operating systems so we don't use them here.)

More precisely, state that changes in time can be modelled in two ways:

1. **Imperatively**, as a pair or tuple consisting of the value of the program counter and the set of all the program's constant and mutable variables together with their current values. (Mutable variables are those that hold different values at different execution times by means of assignments.) For example, in an imperative program with only two variables x and y which suffer a series of direct or indirect assignments during execution, the state at time t would be the tuple

   $$\langle\, t\ ,\ \{\langle x, v_x\rangle, \langle y, v_y\rangle\}\ \rangle$$

   where v_x is the value stored in x, and v_y the value stored in y, at time t. Execution is measured in terms of number of executed instructions so t is the value of the program counter.

2. **Purely functionally**, as the sequence of immutable state values that are passed and returned from function to function. Take for instance the assignment $x := w$. Supposing the assignment takes place at time t and requires k instructions, the change of state can be depicted functionally (assume v_w is the value of expression w):

   $$\langle\, t\ ,\ \{\langle x, v_x\rangle, \langle y, v_y\rangle\}\ \rangle \quad \rightarrow \quad \langle\, t+k\ ,\ \{\langle x, v_w\rangle, \langle y, v_y\rangle\}\ \rangle$$

   Let's name the state to the left of the arrow s_0 and the state to the right of the arrow s_1. The assignment command can be modelled by the following function

   $$\text{update}\ \langle x, w\rangle\ s_0\ =\ s_1$$

   This function takes the variable and expression involved in the assignment plus an initial state, and returns a *new* state. In a functional setting, the value of variable s_0 holding the state before the assignment is immutable amd so the value of s_1.

The difference between the two approaches is clearly underlined by the *naming* or *referencing* scheme. In an imperative program the same variable *name* can refer to different *values* at different execution times, and thus we distinguish between *L-values* and *R-values*. This is not the case in a purely functional program, where a name is just a label for a value; in other words, names are *persistent*, they stand for *R-values*.

The key in modelling stateful computation in purely functional terms is thus to pass around the state as an extra argument from function to function. Take for example the following input-output function

```
getChar :: File → Char
```

that reads a character from a file. If used more than once in the program then it cannot be pure, for it returns a different character depending on the state of the file, a state that changes during the execution of the program—*e.g.*, it changes after calling `getChar` as the disk head has advanced one position.

The functional solution is to pass the whole state of the program, which includes the state of the file, as an extra argument to the function, and make the function return the new state together with the value computed.

```
getChar :: File → State → (Char,State)
```

The state must be passed on from function to function but, unlike arbitrary values, the state should not be copied or constructed anew from previous state values or the scheme would be infeasible. A program state

may contain zillions of components! It has to be somehow sneakily updated destructively in an imperative fashion by the run-time system.

For this to be possible, only one state value can be manipulated or updated at any given time. In other words, a state variable cannot be shared amongst expressions and all computation involving states must be serialised.[2] For example, the following does not make much sense:

```
let (c1,s1) = getChar file s0
    (c2,s2) = getChar file s0
in...
```

The two call expressions to **`getChar`** share state `s0`, and the second call reads a character from the initial state oblivious to fact that a character has already been read from the file.

The composition of two impure unary functions must proceed sequentially with respect to the state:

```
g :: t0 → State → (t1, State)
f :: t1 → State → (t2, State)

compose f g v0 s0 = let (v1,s1) = g v0 s0
                        (v2,s2) = f v1 s1
                    in  (v2,s2)
```

Function `g` takes a value of type `t0`, a value of type `State`, and returns a tuple whose first component is of type `t1` and whose second component is the new value of type `State`. Function `f` takes a value of type `t1`, returns a value of type `t2` and also modifies the state. As the code for `compose` shows, the state has to be passed around sequentially and explicitly: first, `g` is applied to `v1` and state `s0`, producing the value `v2` and state `s1`, and then `f` is applied to `v1` in state `s1`. This is not only tedious but error-prone.

MONADS

Monads make the state implicit and hidden by wrapping the type of functions with side-effects—*i.e.*, the type of functions that take the state and return a value along with the updated state—into an *abstract data type*. State manipulation takes place only via two operations, one called `thenM` that serialises the computation with the state and another called `returnM` that recovers the value computed. (Some people use the names *bind* and *return*).

More precisely, function `thenM` takes two arguments: a stateful computation, *i.e.*, an expression where an impure function is applied to its value arguments but not yet to the current state, and an impure function. The stateful computation is applied by `thenM` to the current state producing a new value and a new state, which are in turn passed to the impure function, producing a final value-state pair result. The following code shows the precise definition—read `t` as 'value-type' in type signatures, and read `v` as 'value' and `s` as 'state value' in function definitions. Type `State` is the type of state values that can be managed internally by the run-time system. In many implementations, what is actually passed around as a state value is a pointer to a global variable. The definition of type synonym `M` can be read as "the type `M t` is the type of functions that change the state and return a value of type `t`":

```
type M t = State → (t,State)

thenM :: M t0 → (t0 → M t1) → M t1

h `thenM` f = λs → let (v1,s1) = h s
                   in f v1 s1
```

Function `returnM` takes a pure value and returns a function that given a state, returns that very value and

[2] This is not entirely correct. Two state values may coexist as long as they are kept apart and don't interfere, that is, as long as no impure function ever works with both. Think of them as the states of two completely independent execution threads.

the state unchanged—*i.e.*, if `v` is a value, the application `returnM v` is a stateful computation that returns that value without affecting the state; hence, `thenM` is the sinful operation that allows us to turn pure values (which live in a stateless world) into impure ones (which live in a stateful world). And once you go impure, there is no possible redemption: we have not defined an operation that takes a stateful computation and gets the value and ignores the state; `thenM` passes values and states from a stateful computation to an *impure* function. The definition of `returnM` is:

```
returnM :: t → M t
returnM = λv → (λs → (v,s))
```

Impure function composition can be defined in terms of these operations as follows (the inverse quotes are Haskell's syntax for writing binary functions in infix form) :

```
compose f g v0 s0 =
    (g v0) `thenM` (λv1 → (f v1) `thenM` (λv2 → returnM  v2)) s0
```

here, the impure function `g` is applied to `v0` in the initial state `s0`, and the value it computes, `v1`, is passed by `thenM` to its second argument, which is the (nameless) function

```
λv1 → (f v1) `thenM` (λv2 → returnM  v2)
```

that is, a function that takes a value `v1` and calls `thenM` again but feeding `v1` to `f`—which produces a value that `thenM` passes to a second nameless function that calls `returnM` to recover the value computed. The state is implicitly passed sequentially from one impure function to the next, *i.e.*, from `g` to `f`. Compose would have to execute in an initial state but it can be passed automatically by the run-time system. Indeed, we could have omitted the initial state `s0` in the definition of compose:

```
compose f g v0 =
    (g v0) `thenM` (λv1 → (f v1) `thenM` (λv2 → returnM  v2))
```

This program can be made to look remarkably imperative if written using the syntactic sugar known as the *do-notation*. The actions (computations) within a **`do`** take place sequentially.

```
compose f g v0 = do v1 ← g v0    -- 'thenM' implicit here
                    v2 ← f v1    -- 'thenM' implicit here
                    returnM v2
```

The state is finally encapsulated when the polymorphic type `M` is defined as an *abstract data type* instead of as a type synonym. The type of **`getChar`** is arrived at as follows. Initially:

```
getChar :: File → State → (Char,State)
```

Using abstract data type `M`:

```
getChar :: File → M Char
```

In Haskell, for the case of input-output, type `M` is called **`IO`**; therefore:

```
getChar :: File → IO Char
```

Figure 1 shows an example of usage: the impure function `getString` reads a string (a sequence of characters) from a file. The first box shows the function written in terms of `thenM` and `returnM`. The second box is a rendering of the first box in much more readable *do*-notation, which is also more imperative in style. Notice the use of recursion in both.

UNIQUE TYPES

Unique types rely on the uniqueness of values to combine purity and effects. The idea is to signal to the type checker when arbitrary values, not just state values, are not shared. Take for example the input-output

```
getString :: File → IO String

getString file =
  (getChar file) ‘thenM‘
     (λc → if c == EOF then (returnM [])
                       else ((getString file) ‘thenM‘ (λs → returnM (c:s))))
```

```
  getString file =  do c ← getChar file
                       if c == EOF then returnM []
                                   else do s ← getString file
                                           returnM (c:s)
```

Figure 1: Impure function `getString` reads a string from a file.

function **`putChar`** that writes a character to a file.

```
  putChar :: Char → File → File
```

If its file argument is not shared by any other expression there is the guarantee that it will not be used by the program after the function call and therefore it can be garbage-collected. However, instead of constructing a new file value, we can destructively update the old one and return it again re-using its memory space. Uniqueness is enforced by the type checker based on programmer's annotations. We would write the type of the function as follows

```
  putChar :: Char → *File → *File
```

where the star signals to the type checker that the value passed as a second argument to **`putChar`** must not be shared by two or more expressions and that it must enforce the same property for the value returned.

The state is also hidden, but unique types impose a form of serialisation and sharing discipline on the programmer which must be aware of these constraints when writing the program and make sure they satisfy the type checker. The sharing discipline becomes straightforward when expressions are represented directly as *graphs*. Unique types are related to *linear types* (and consequently to *linear logic* due to the Curry-Howard Isomorphism that identifies types with formuli in constructive logics and functional programs with proofs); unique types provide information about the way a specific function is applied (in a linear way, never shared) while linear types provide information about the way an expression is used *within* a function.

SUGGESTED READING

We have only glossed over the bare basics of monads and unique types from a programming perspective and particularly in relation to input-output. The following are some further references:

- The short online tutorial *What the Hell are Monads?* by Noel Winstanley is a good starting point. It can be found at

  `http://www.bercrombiegroup.co.uk/~noel/uploads/research/monads.html`

- Philip Wadler wrote a series of articles that popularised monads inside and outside the functional programming community. They can be found on his web site:

  `http://homepages.inf.ed.ac.uk/wadler/topics/monads.html`

- The tutorial article by Meijer and Jeuring, *Merging Monads and Folds for Functional Programming*, published in [JM95], shows how monads integrate swiftly with classical functional programming assets such as recursion patterns and higher-order functions. They present examples of what are nowadays called *monadic folds*.

- The title of Simon Peyton Jones' tutorial says it all: *Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-language Calls in Haskell*. It can be found at `http://research.microsoft.com/~simonpj/papers/marktoberdorf`
- Many other articles and introductions about monads can be found in the Haskell web site:

  `http://www.haskell.org/bookshelf/#monads`
- Unique types are heavily employed by the functional programming language Clean, especially in its Object-I/O library. The book [PVv93] is a good introduction to Clean and unique types. More information is available at the Clean web site `http://www.cs.kun.nl/~clean`

MONADS AND CATEGORY THEORY

The concept of a monad is slightly more elaborate than we have described. It originated from Category Theory, an abstract branch of mathematics that deals with abstract structures and the relationships between them. According to folklore, the relation between monads and computation was discovered by Eugenio Moggi [Mog91] and then popularised and applied to functional programming by Philip Wadler. Moggi proposed the category-theoretic notion of monad as a means of parameterising programming language semantics by a stateful computation that produces a value—in short, to avoid passing the state explicitly from semantic function to semantic function and have it passed internally, as explained above.

The term 'monad' has nothing to do with Leibniz's monads (sort of spiritual or mental atoms) but was formed by combining **mon**oid and tri**ad** (*i.e.*, triple). In Algebra, a set of elements for which there is a binary operation with identity and associativity is called a monoid. In Category Theory, a triple consists of a *functor* and two *natural transformations* which satisfy certain properties. In our description, a functor corresponds to the type hiding the state and the natural transformations to the parametrically polymorphic operations `themM` and `returnM`. The combination of the two terms into the term 'monad' is suggested by the fact that the monoid operation is composition in the particular category.

Category theory is heavily used in programming language theory, especially in denotational semantics, algebraic specification, and program construction. The central concepts of these disciplines are usually wielded in their categorical formulation.

To give a rough idea, a category is a collection of 'structure-preserving' mappings (called arrows) between certain 'sets with structure' (called objects). Arrows are closed under composition, which must be associative, and there is an identity arrow for every object. These axioms describing what constitutes a category are rather general and wildly different mathematical structures can be 'categorised as categories'.

Many important properties of arrows do not depend on the particular structures under consideration and can be studied abstractly and independently of them. These *universal properties* are expressible *externally*, that is, purely in terms of composition of arrows. Unlike, for example, set theory, where we are concerned with the internal structure of sets and mappings, the 'categorical' approach abstracts away from this detail and concentrates on the external relationships between the sets. For example, in set theory injective functions are characterised in terms of a property held by the elements of their domain and codomain sets, whereas from a categorical perspective the equivalent concept (monic arrows) is defined in terms of the properties under composition of these mappings.

Category theory gets really interesting when we start considering structure-preserving arrows between categories, called *functors*, and structure-preserving arrows between functors, called *natural transformations*. In programming terms, the category of interest is the category of types, where objects are non-parametric or *manifest* types and arrows are monomorphic functions between them. Functors are parametrically polymorphic *type operators* and their **`map`** operation; natural transformations are polymorphic functions between such type operators. For example, the type **`List`** is a functor. It is a type operator taking a manifest type `T` to the manifest type **`List`** `T` and has a map operation

```
mapList :: (a → b) → List a → List b
```

The type signature tells us that given a function (arrow) `f` from manifest type (object) `a` to manifest type (object) `b`, then `mapList f` produces a function (arrow) from the manifest type (object) obtained after applying functor **`List`** to `a`, to the manifest type (object) obtained after applying functor **`List`** to `b`.

The polymorphic function **`reverse`** that reverses a list is a typical example of a natural transformation:

```
reverse :: List a → List a
```

The reason for this is the equation

```
reverse ∘ (mapList f) == (mapList f) ∘ reverse
```

where function `f` has type `a→b`. The equation tells us that if we map a list of type **`List`** `a` into a list of type **`List`** `b` and then reverse the latter, we get the same result by first reversing the original list and then mapping it afterwards. This is because **`reverse`** is a natural transformation. In general, a natural transformation η is an arrow between two functors F and G, such that for any two objects a and b of the category, the following equation holds

$$\eta_b \circ (mapF\ f) = (mapG\ f) \circ \eta_a$$

where

$$\begin{aligned} \eta_a &:: F\ a \to G\ a \\ \eta_b &:: F\ b \to G\ b \\ mapF\ f &:: F\ a \to F\ b \\ mapG\ f &:: G\ a \to G\ b \end{aligned}$$

In our example with reverse, $F = G =$ **`List`**.

It is common practice to blur the distinction between *mapF* and F (*e.g.*, to use the same name **`List`** for both the type operator and the map function on lists) and write the equation as follows:

$$\eta_b \circ (F\ f) = (G\ f) \circ \eta_a$$

Things don't stop here. Special objects like *products*, *limits*, *pullbacks*, etc, and their duals all have a neat programming interpretation. They shed light on the semantics of programs and help in guiding language design, program construction, program proving, and program transformation.

SUGGESTED READING IN CATEGORY THEORY

There are many sources on category theory addressed specifically to computer scientists. Here are some suggestions:

- The vintage, out-of-print book by Robert Goldblatt [Gol79] deals mostly with Categorical Logic, but the first chapters present the central concepts of Category Theory inductively from Set Theory. This is an asset and a hazard. An asset because most concepts are abstracted away from set-theoretical intuitions. A hazard because the reader might not 'abstract' enough and could become too tied to the set-theoretical motivation.
- Joseph Goguen's *Categorical Manifesto* [Gog91] explains why category theory is useful in Computing Science.

- Benjamin Pierce's book [Pie91] is often recommended, but more than a textbook it is a short compendium of definitions and examples, many of which are taken from [Gol79]. Solutions are not provided for most exercises. It has an excellent annotated bibliography.

- Most texts in category theory indulge in their own verbose proof style. An exception is Fokkinga's treatment of category theory from a calculational standpoint [Fok92]. The calculational style uniformly permeates the definitions, theorems and proofs. This style of presentation is very common amongst the *squiggol* community of functional programmers interested in program construction by program transformation.

- A fresh approach to the topic can be found in *Category Theory as Coherently Constructive Lattice Theory* by Backhouse, Bijsterveld, van Geldrop, and van der Woude. This tutorial draws on Lattice Theory as the intuitive ground from which to grasp the abstract notions and definitions of category theory, and it quite demystifies the subject. The paper can be found at

  `http://www.cs.nott.ac.uk/~rcb/papers/abstract.html#CatTheory`

- The first chapters of *The Algebra of Programming* book [BdM97] introduce categories in the context of types, functional programming, and their use in program construction and transformation. By the way it is written, it presupposes some acquaintance with the basic intuitions.

- Walter's textbook [Wal91] focuses on the algebra of functions and has a chapter devoted to the categorical description of data types.

- Chris Hillman's [Hil01] and Barry Jay's [Jay] tutorials are not specifically addressed to computer scientists, but they have a wealth of examples taken from varied mathematical disciplines.

- The often-cited classic, and hard, references are [AL88], [BW99], and of course [Mac71].

*

References

[AL88] Andrea Asperti and Giuseppe Longo. Categories, types, and structures: an introduction to category theory for the working computer scientist. Electronic book, 1988.

[BdM97] Richard Bird and Oege de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice-Hall, 1997.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. International Series in Computer Science. Prentice-Hall, 1988.

[BW99] Michael Barr and Charles Wells. Category theory. Lecture Notes, ESSLLI, 1999.

[Fok92] Maarten M. Fokkinga. A gentle introduction to category theory — the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, pages 1–72 of Part 1. University of Utrecht, September 1992.

[Gog91] Joseph A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, March 1991.

[Gol79] Robert Goldblatt. *Topoi: The Categorial Analysis of Logic*. North-Holland, New York, 1979. Out of print.

[Hil01] Chris Hillman. A categorical primer. Electronic print, 2 July 2001.

[Hug89] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[Jay] C. Barry Jay. An introduction to categories in computing. Electronic document.

[JM95] Johan Jeuring and Erik Meijer, editors. *1st International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995*, volume 925 of *Lecture Notes in Computer Science*. Springer, 1995.

[Lev04] Paul Blain Levy. *Call-By-Push-Value. A Functional/Imperative Synthesis*. Kluwer Academic Publishers, Dordrecht, 2004.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[PVv93] Rinus Plasmeijer, Marko Van Eekelen, and Marco van Ekelen. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.

[Rea89] Chris Reade. *Elements of Functional Programming*. International Series in Computer Science. Addison-Wesley, 1989.

[Wal91] R. F. C. Walters. *Categories and Computer Science*. Number 28 in Cambridge Computer Science Texts. Cambridge University Press, 1991.